PERFORMANCE EVALUATION OF REAL-TIME SCHEDULING ON A MULTICOMPUTER ARCHITECTURE

*+Luis F. Friedrich, +Rafael Cancian, *Rômulo S. de Oliveira. +Thadeu B. Corso*

+ Departamento de Informática e de Estatística
* Departamento de Automação e Sistemas
Universidade Federal de Santa Catarina
CP 476 Florianópolis SC Brasil - CEP 88040-900
+{lff, cancian, thadeu}@inf.ufsc.br, *romulo@das.ufsc.br

ABSTRACT

The complexity of some real-time applications demands high performance computer architectures. Multicomputer architectures have a potential for high performance and reliability because of their expressive number of processors and communication channels. Therefore, they are natural candidates for supporting complex real-time computing. This paper presents a performance evaluation of real-time scheduling in a parallel/distributed environment which is based on a multicomputer architecture.

I. INTRODUCTION

Real-time systems are evolving from simple applications to more sophisticated ones with stringent performance and reliability requirements. Many applications in the real-time domain cannot meet their requirements without highly cooperative computer systems consisting of multiple computing nodes providing features such as parallel computation, scalable performance growth, and fault tolerance. Applications in areas such as robotics, and real time simulations not only have time requirements but they also demand high processing capacity.

Multicomputer systems are natural candidates for the most demanding real-time applications because of their potential for high performance and reliability. Multicomputers designate machines consisting of a set of nodes, which are linked by an interconnection network. Each node is minimally constituted by a processor, dedicated memory and physical communication channels. The presence of multiple physical channels between nodes in the interconnection network makes the system robust to connection and node failures. It also makes possible to support simultaneous transmissions on different communication channels. Multicomputers differ from each other basically on the topology of their interconnection networks, which can be classified as static or dynamic (Feng 1981). In a static interconnection network the links between nodes are passive and can not be reconfigured to establish connections between other two nodes, physical channels are permanent. On the other hand, links in a dynamic interconnection network can be reconfigured through switching circuits which manipulation provides the assignment of temporary physical channels.

The construction of real-time applications as *communicating process networks* is similar to multitasking in a centralized single processor or multiprocessor but without shared (real or virtual) memory. A real-time application is written as a set of components (*tasks*) that interact with each other to perform the application. This interaction is logically independent of which component resides on which computing node. The tasks cooperate based on message passing through logical channels. A communicating process network denoting a real-time application can be expressed as a graph in which vertices represent tasks and edges represent communication channels. The resulting graph may represent real-time applications either with a static topology during its entire execution or, with a dynamic topology resulting from dynamic creation of tasks and logical channels (resources).

This paper focuses on the performance evaluation of real-time scheduling solutions in a parallel execution environment which is based on a multicomputer architecture. The motivation is based on the fact that these machines are a natural choice as execution environment for real-time applications, especially those requiring high performance. The paper is composed of the following sections: section 2 describes the architecture of the CRUX environment, section 3 describes the operation mode of the execution environment, section 4 describes the programming model and the scheduling approach used, section 5 discusses the simulation results and section 6 presents the final considerations.

II. THE CRUX ARCHITECTURE

The general model of the CRUX environment is based on a multicomputer architecture composed by a set of processing nodes interconnected by a crossbar network (Corso and Fraga 1998), as shown in figure 1.

The CRUX multicomputer-based architecture is made of

- A number of *worker nodes* (WN), each one contains a processor with dedicated memory and a communication hardware which provides communication channels for the connections between the WN's.

- A *control node* (CN) to manage the connection/disconnection of physical channels between working nodes (in addition to allocation/liberation of working nodes).
- A *working network* (a crossbar) to transport messages between worker nodes through physical channels operated by the control node.
- A specific control network (a control bus) to convey messages between the worker nodes and the control node.

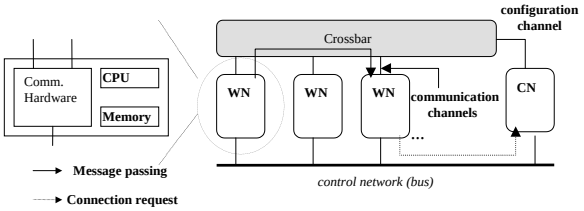

Figure 1: The CRUX architecture.

The interconnection network, in the proposed architecture, is based on a communication mechanism with demand-driven assignment of physical channels composed by two basic levels of communication: *working network* (crossbar) and *control network* (bus). The crossbar, N dimension, is used to transport the messages produced by the application tasks through point to point (direct) channels between the worker nodes in which the tasks are executing. It allows N simultaneous and exclusive connections between peer nodes. The control network is used to deal with the problem of connecting the physical channels of the working network. It transports run-time support messages, between the worker nodes and the control node. The messages carry the requests for connection/disconnection of the working network physical channels. Therefore, before proceeding to the effective communication between two worker nodes, the existence of a (direct) point to point channel between them through the working network is verified. If there is a channel, the communication can be immediately initiated. If not, it must be preceded by a connection procedure.

III. MODE OF OPERATION

The target architecture provides various modes of operation, each one having a different manner of using and sharing resources such as processors and crossbar, and the possible solutions for real-time scheduling on that architecture depends on the mode of operation (Friedrich and Oliveira 1998). In all cases the main resources to be scheduled are the processors, the communication channels and the control network. Other sharing resources such as data structures and devices are not object of this paper. This section describes various modes of operation that are possible in the target architecture resulting in different forms of using and sharing the processors and the crossbar. The control network is dependent on how that two resources are manage.

Regarding the processors, there are two basic modes of operation: single-task and multiple-task:

SINGLE-TASK - In single-task mode each processor executes a single application task (thread of execution). In this case, if the task is not ready for execution the processor is idle.

MULTIPLE-TASK - In multiple-task mode a processor executes one or more tasks and the communication between them is done through shared memory for tasks running on the same node, and through the crossbar for tasks running on different nodes. For remote communications there are two alternatives: (1) there is only one local task which is responsible for sending messages to one destination or, (2) more than one task is allowed to send messages to the same destination.

Regarding the crossbar utilization, we have identified five modes of operation: Static, Routing, Demand, Synchronous and Broadcast:

- STATIC - In this mode the crossbar has a static configuration which is calculated at design time, in order to satisfy all the inter-processor communication needs using point-to-point physical connections. There is no reconfiguration of the communication channels at execution time. Once the static configuration is established the control node is not useful anymore and a fault on that node should not affect the execution of the application. Allocation of processes to processors in this case is straightforward. However, not all requested configurations are possible.
- ROUTING - As in the static mode, the crossbar has a static configuration calculated at design time and established during application's initialization. However, in this mode it is possible to have logical connections, meaning that communication between two processor can be done through a third one which is the routing processor. This mode is more flexible than the static one because it provides a static configuration of the crossbar even when there is no point-to-point physical connection possible between all peers. Some logical connections would share one physical connection. It is assumed that most of the connections would be point-to-point physical connections and that the logical connections would have at most one routing processor.
- DEMAND - When the number of logical connections is high, it could be better to have the connections established by demand. In this case, the crossbar configuration is dynamic, meaning that every time a task has to send a message to another task located in a different work node the sending task needs first to ask for a connection with that node. The requests for connections are executed by the control node which informs the task after the connection was established. In this mode, every logical connection is mapped to a point-to-point physical connection. In other words, there is no routing. After sending the message the physical connection is released. Here, issues like the time for connection establishment and possible

collisions between various requests affect the execution time of the tasks. So, it is necessary to schedule not only the crossbar but also the auxiliary network. This is the natural mode of operation for non real-time tasks.

- SYNCHRONOUS - This mode of operation also provides a dynamic configuration of the crossbar. However, the connections are not established by demand. In this mode, the control node establishes the connections based on a configuration table which is built at design time. The configuration table is cyclic and determines which point-to-point physical connections are to be established at each time period. When a task wants to send a message to a remote one it must wait for the next time period in which the requested point-to-point physical connection will be available. In this mode of operation all the nodes must be clock synchronized and the complexity of the auxiliary network scheduling depend on the synchronization algorithm used.
- BROADCAST - A different approach which could eliminate the need for clock synchronization is the control node to broadcast, through auxiliary network, whenever a point-to-point physical connection is established or released. As in the synchronous mode there is a configuration table which is calculated at design table and states when each physical connection is established and released. The control node executes the configuration table based on its local clock and also broadcasts the events. This way, each work node knows at each time which connections are available for sending messages.

The remainder of this section describes how the target architecture has been used to provide an execution environment for real-time applications.

The system executes a single application, which is composed by many processes. Each WN executes a process that can be composed by one or more tasks. Each task is a scheduling unit. The crossbar configuration is dynamic. Every time a task in one WN needs to send a message to another WN, first it send a message through the control network to the CN. After the physical connection is setup the sending task is warned by the CN and the communication can take place. Every logic communication channel is mapped directly into a physical communication channel, without routing. By the time a task finished sending a message it must release the physical connection.

IV. PROGRAMMING MODEL AND SCHEDULING APPROACH

In this work we are considering soft real-time applications, in which a deadline miss will not have catastrophic consequences. However, it is supposed that the quality of the application is proportional to the number of deadlines achieved. The applications are developed following the client/server programming model. This programming model results in graphs with linear precedence. The client is divided in two parts: before calling the server and after calling the server. Therefore we have a linear graph composed by 3 tasks: (1) client before call, (2) server, and (3) client after call. Figure 2 illustrates the situation. In addition, a server might be client of another server and so on. In this case we would have a graph composed by 5, 7 or more tasks.

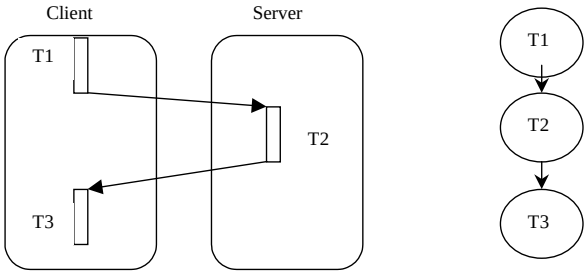

Figure 2: Client-Server relationship translated to a precedence graph.

An activity is defined as a set of tasks with precedence relationship, direct or indirect. The execution of an activity is initiated by some event happening in the environment, such as the activation of a sensor or timeout signal. The result of the activity is a response from the system to the environment. This response is in part a function of the event that was observed. That way, deadlines are not individually associated with tasks, but they are associated with all the activity. Therefore, an activity accomplishes its deadline only when all tasks of the activity finish up to the deadline.

It is supposed that activities are aperiodic and we must apply a best effort scheduling approach. It is also supposed that tasks were previously allocated to the processors and there is no task migration during the execution.

The scheduling solution under evaluation in this paper applies EDF (Earliest Deadline First) [Liu and Layland 1973) to schedule the tasks on each processor. Each task has an associated deadline that is the same deadline associated with the task activity. This solution is easy to implement and presents a low overhead.

As described in section 3, when a task needs to send a message to another task residing in a different processor it must request a crossbar connection to the CN. If the number of connection requests is greater than the crossbar capacity (number of physical channels) then we have a queue of requests that must be kept by the CN. In this case, it is necessary to decide which algorithm will be used by the CN in order to attend the queue of requests. Although a FCFS (First-Come First-Serviced) algorithm seems to be the fairest one and is probably the right choice for a conventional system, the goal of a real-time system is to maximize the number of deadlines achieved.

In this paper we evaluate, through simulation, the behavior of the following 3 different algorithms for attending connection requests:

- FCFS (First Come First Served), the scheduling of the requests is done in order of arrival;

- EDF (Earliest Deadline First), the scheduling is done based on the deadline of the activity associated with the message;
- SMF (Smallest Message First), the scheduling is done based on the message size.

V. PERFORMANCE EVALUATION

The performance of the proposed communication mechanism for real-time applications was evaluated by simulation in a way similar to what was done in (Friedrich et al. 1998) for another task model and scheduling approach. It was supposed in the simulation that the implementation of the communication mechanism, which is based on a demand-driven assignment of physical channels, is performed by the run-time support composed by a central component (executed in the CN) and local components (executed in each WN). The Control Node is responsible for the crossbar operation, executing the connection and disconnection of physical channels.

A set of applications was randomly generated. Three parameters varied during the simulations:

- The algorithm used by the CN, with three possibilities: FCFS, EDF, SMF.
- The number of Worker Nodes, with four possibilities: 8, 16, 24, 32.
- The number of physical channels in each WN, with two possibilities: 1, 2.

The model used to simulate the system was created using the Arena 3.01 system simulation software. The model simulates the main behavior of the environment, composed by the Control Node (CN), the Worker Nodes (WN), the Working Network and the Control Network.

Activities are composed by tasks that run on WN processors. After first executing on a client processor, tasks may communicate with another WN by sending a message through the crossbar. To send a message the task must request to the CN a connection with the other WN. To make that request, tasks use the Control Network.

When a message arrives on the destination WN (server), a new task (belonging to the same activity) is created and putted in the ready queue of that processor. After getting the processor and finishing the execution, another message is generated and sent back to the client WN following the same procedure. When it arrives at the client WN, other task executes (task T3 in figure 2).

In each experiment the model simulates the execution of 1000 activities. During this time it collects statistics about the system behavior. Each activity has a deadline and all tasks that belong to the activity share this value. The model chooses randomly a WN to start its execution. There is no workload balance.

The number of WNs varies with the simulations since it is a parameter to be analyzed. It assumes the values 8, 16, 24 or 32 (*vMaxNodos*) in the simulations done.

An activity is formed by tasks, that are the real schedulable entities. A task executes in a WN processor for 20 units of time (*vTmpProcNT*). The WN uses a preemptive EDF scheduling algorithm for the processor, with a quantum of 1 (one) unit of time (u.t.) (*vQuantum*) for tasks with the same deadline.

After executing in a WN processor, the task may either send a message through the crossbar to another WN or just finish (finishing the whole activity) if there is no more communication to do. The number of messages sent by an activity is 4 (*vNumMensagens*), what means the creation of five tasks (T1, T2, T3 in figure 2 and two more tasks, T4 and T5). Three of them executing on the client, and two executing on the server(s). There may be more than one task trying to execute on the same WN processor at the same time.

The model separates some WN to create a server pool while the others WNs act only as clients. When a client WN sends a message, it chooses randomly one WN from the server pool. When a server WN sends a message it always send it back to the same client WN that caused the task execution on the server.

To send a message, the task must request a connection to the Control Node (CN) through the Control Network. The period for pooling the control bus is 0.1 u.t. (*vTmpFreq*) and the execution time of each request on the CN is 0.5 u.t. (*vTmpProcNC*). Three algorithms were used by the CN to schedule the requests: FCFS, EDF and SMF.

The size of the messages varies from 2^8 = 256 bytes (*vTamMensMin*) through 2^{12} = 4 Kb (*vTamMensMax*), following an uniform distribution. After the connection has been established by the CN, the message is transferred through the crossbar. The time to send a 1 Kbyte message through the crossbar is 20 u.t. (*vTmpComm*).

Each WN has one or two physical channels (*vMaxCanais*) to send messages. To evaluate the impact of the number of channels for each WN, two groups of charts are shown. The first one shows the behavior of the algorithms with one physical channel, and the second group uses two physical channels for each WN.

The CN allocates just the necessary number of channels to meet the message deadline. Once a connection is requested the CN verifies the availability of the communication channels on both origin and destination Worker Nodes and also verifies how many channels (*aNumCanais*) is necessary to send the message in order to meet the deadline. If there are enough physical channels available then the communication is established and the communication time is calculated as a function of message size (*aTamMensagem*), communication delay (*vTmpComm*) and number of physical channels used (*aNumCanais*). If the available communication channels are not enough to meet the message deadline, the available number of channels are allocated and the communication is established anyway. If there is no available channels then the request stays in the CN queue for scheduling. The message may miss its deadline but it is never discarded.

The deadline chosen for an activity varies from a minimum value (*eTmpDeadMin*) to a maximum value (*eTmpDeadMax*) expressed as follow:

$$eTmpDeadMin = vTmpProcNT + vTmpFreq + vTmpProcNC + vTmpComm * (2^{vTamMensMax}) / 1024$$

$$eTmpDeadMax = 2 * vTmpProcNT + (vMaxNodos * vTmpFreq) + (vMaxNodos * vMaxCanais * vTmpProcNC) + vTmpComm * (2^{vTamMensMax}) / 1024$$

Value *eTmpDeadMin* is the minimum time needed by a task to execute on a WN and to send the biggest message without any delay. It consists of the execution time on the WN (*vTmpProcNT*), time waiting in the Control Network (*vTmpFreq*), time executing on the CN (*vTmpProcNC*) and time to send a $2^{vTamMensMax}$ Kbyte message.

The *eTmpDeadMax* is that time assuming maximum delay in all stages but no more than 2 tasks concurring for the WN processor. The deadline (*aDeadline*) of an activity follows a triangular distribution with these parameters:

$$TRIA(eTmpDeadMin, eTmpDeadMin + (eTmpDeadMax - eTmpDeadMin)/3, eTmpDeadMax).$$

The time between arrivals for activities, the activity period, is

$$ePeriodoChegada = mpDeadMin * 0.7$$

and is constant for all simulations. If a task misses its deadline while executing or sending a message, it is not deleted but it is marked as missed and continues normally. Just when all tasks belonging to an activity finish then the activity is classified. If at least one of its tasks missed its deadline, then it is considered that the whole activity missed the deadline. Otherwise the activity met the deadline.

Each one of the three CN's scheduling algorithms were simulated 5 times with 8, 16, 24 and 32 WNs executing 1000 activities, for a total of 60 simulations (and 300000 tasks executed). For each group of 5 simulations it was calculated an average. These average simulations of the 3 algorithms with 8, 16, 24 and 32 WNs are showed bellow, in two groups: with one or two physical channels for each WN.

Figure 3 shows how much time (*ut*), in average, the tasks spend in getting connected. Generally, if the communication channel is available the connection time is less than 1*ut*. However, when the solicited channel is not available than the task needs to wait for the channel to be released. The graphics show that the performance of the EDF algorithm is very close to the FCFS.

Figure 4 shows the percentage of tasks that missed their deadlines. Deadlines are initially associated to activities which can be divided in more than one tasks. Each task of a specific activities receives as deadline the same deadline of the activity. If a task belonging to an activity miss its deadline the activity also miss the deadline. The graphics show that when using 1 communication channel the performance of the algorithms is poor, except for the SMF. However, with 2 communication channels all three algorithms have a good performance. The use of a crossbar gives a high traffic capability to the interconnection network, similar to N dedicated buses. This capability is multiplied every time we add a new physical channel.

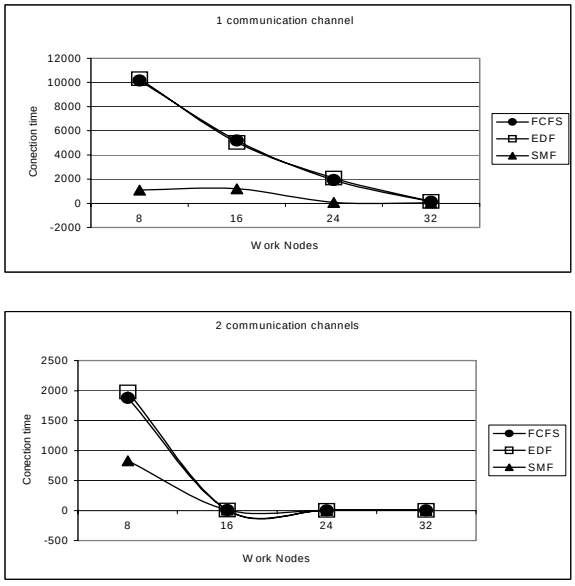

Figure 3: Average connection time of all tasks

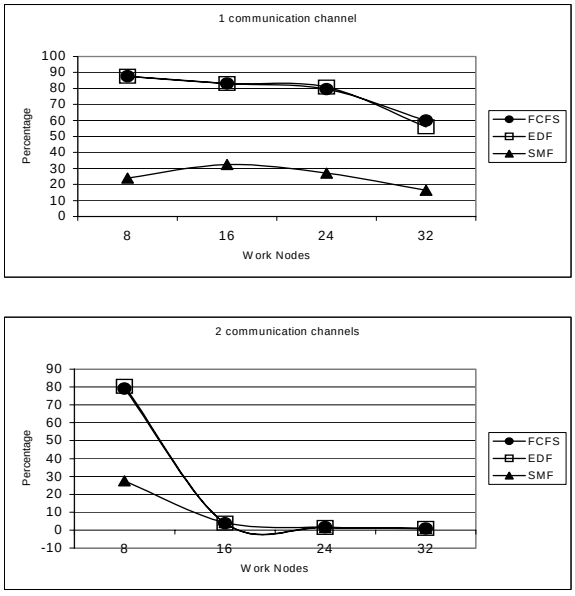

Figure 4: Percentage of tasks that missed deadlines

Figure 5 shows the minimum delay of tasks that missed their deadlines. In this case the delay is calculated by the following:

$$Delay = tnow - (aChegadaTarefa + aDeadline),$$

where *tnow* is the actual time, *aChegadaTarefa* is the task arrival time and *aDeadline* is the deadline. The graphics show that the algorithm SMF has the best performance in both cases (1 or 2 channels). However, while increasing the number of work nodes FCFS and EDF tend to have the same performance of the SMF.

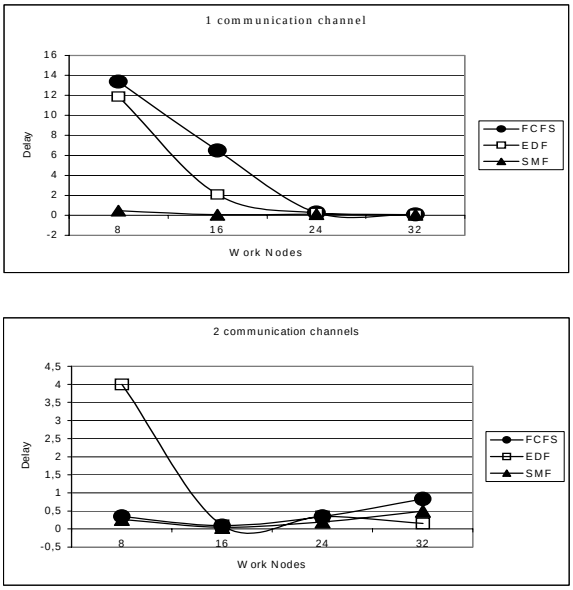

Figure 5: Minimum delay of the tasks that missed the deadline

VI. CONCLUSIONS

With the high transmission rate provided by the crossbar interconnection network, messages are mostly sent without queue formation, what means that the schedule algorithm has a minor effect on the performance under normal workload. Under overloaded conditions, the choose of a scheduler for the CN may produce significant impact on the target architecture. Therefore, in order to evaluate the impact of the scheduler algorithms used by the CN, some factors in the model were aggravated to cause crossbar saturation, when the schedule effects are best shown.

The results show that algorithm SMF causes less deadline misses and produces a minor delay in almost all cases. When the transmission time is significant, small messages first transmits quickly and those short messages will probably meet deadline (best minimum delay). Large messages that would waste much time transmitting and keeping physical channels busy, are scheduled later, producing the worst maximum delay and also worst average delay with two physical channels.

The algorithms FCFS and EDF produced results not as good as SMF, in terms of deadline misses and minimum delay. Their results showed approximately the same values. Because we are simulating soft real-time applications, the tasks are not discarded when they miss a deadline, but continue in the system until all the activity is executed. This mean that old tasks that had missed deadline or will certainly miss it, are scheduled first, just like algorithm FCFS, with the difference that the ordering is defined by the deadline. In hard real-time applications, where tasks are discarded as soon as they miss a deadline, the scheduler becomes free to choose a task that may still meet deadline, and in that case EDF shows a better result, but that is not the case in this study.

According to the results it is worth considering, for the target architecture and the suggested programming model , a real-time scheduling solution in which not only the deadlines are taken into account but also the size of the messages the tasks use to communicate each other.

REFERENCES

Feng, T.-Y., *A Survey of Interconnection Networks*, IEEE Computer, v. 12, n. 14, p. 12-27, December 1981.

Corso T. B., Fraga J. S. , Freitas F. P.. *Demand-Driven Multicomputer environment: Design and Evaluation*, SCS Western Multi Conference 1998, San Diego, USA, January 1998.

Friedrich L. F., Oliveira R. S., Corso T. B. *Escalonamento Tempo Real em uma Arquitetura Baseada em Multicomputadores*, I Workshop de Sistemas de Tempo Real, Rio de Janeiro, Brazil, Maio de 1998. (in Portuguese)

Xu J., Parnas D. L.. *On Satisfying Timing Constraints in Hard Real Time Systems*, IEEE Transactions on Software Engineering, v. 19, n. 1, p. 70-84, January 1993.

Sha L., Rajkumar R., Lehoczky J. P.. *Priority Inheritance Protocols: An Approach to RealTime Synchronization*, IEEE Transactions on Computers, v. 39, n. 9, p. 1175-1185, september 1990.

Anderson J. H., Ramamurthy S. *A Framework for Implementing Objects and Scheduling Tasks in Lock-Free Real Time Systems*, Proc. of the l7th IEEE Real-Time Systems Symposium, december 1996.

Liu C. L., Layland J. W.. *Scheduling algorithms for multiprogramming in a hard real time envirorement*, Journal of the Association for Computing Machinery, v. 20, n. l, January 1973.

Friedrich, Luis F. et al. *On the Performance of Real Time Communication in Multicomputer Interconnection Networks*. 1998 Summer Computer Simulation Conference (SPECTS'98), Reno, Nevada, USA, July, 1998.